An Empirical Comparison of Methods for Iceberg-CUBE Construction

Leah Findlater and Howard J. Hamilton

Technical Report CS-2000-06
August, 2000

Department of Computer Science
University of Regina
Regina, Saskatchewan, CANADA
S4S 0A2

ISSN 0828-3494
ISBN 0-7731-0410-0

An Empirical Comparison of Methods for Iceberg-CUBE Construction

Leah Findlater and Howard J. Hamilton

Department of Computer Science
University of Regina
Regina, SK, Canada S4S 0A2
{findlatl, hamilton}@cs.uregina.ca

Abstract

The Iceberg-Cube problem is to apply an aggregate function over a set of attributes to determine which combinations of attribute values are above a specified aggregate threshold. We implemented bottom-up and top-down methods for this problem. The bottom-down method we used already used pruning. Results show that even when the top-down method employed pruning, it was slower than the bottom-up method because it did not prune as efficiently.

1. Introduction

Users of decision support systems generally see data in the form of ***data cubes*** [5]. The cube is used as a way of representing the data along some measure of interest [4]. It can be two-dimensional, three-dimensional or higher-dimensional. The cells in the data cube represent the measure of interest, for example they could contain a count for the number of times that combinations occurs. Suppose we have a database which contains transaction information that represents sales of a part from a supplier to a customer [4]. Each cell (*p*,*s*,*c*) in the cube part-supplier-customer represents the total sales of part *p* to customer *c* from supplier *s*. With three attributes we have eight possible combinations: part; customer; supplier; part, customer; part, supplier; customer, supplier; part, customer, supplier; and none (no attributes). In the context of database retrieval these combinations are called ***group-bys***.

Queries are performed on these cubes to retrieve decision support information. The goal is to retrieve data in the most efficient way possible. Three possible solutions to this problem are to pre-compute all cells in the cube, to pre-compute no cells, and to pre-compute some cells. The size of the cube for attributes $A_1,...,A_n$ with cardinalities $|A_1|,...,|A_n|$ is $\pi|A_i|$. Thus, the size increases exponentially with the number of attributes and linearly with the cardinalities of those attributes. To avoid the memory requirements of pre-computing the whole cube and the long query times of pre-computing none of the cube, most decision support systems pre-compute some cells.

The ***Iceberg-Cube problem*** is to compute only those group-bys or combinations of attribute values that satisfy a minimum support requirement or other aggregate condition, such as average, min, max, or sum [2]. The term "iceberg" was selected because these queries retrieve a relatively small amount of the data in the cube, i.e. the "tip of the iceberg" [2]. Two straightforward approaches to this problem are top-down and bottom-up. The bottom-up approach starts with the smallest, most aggregated group-bys and works up to the largest, least aggregated group-bys. The

top-down approach starts with the largest, least aggregated group-bys and works down to the smallest, most aggregated group-bys.

In this paper, a comparison of the efficiency of bottom-up and top-down methods is provided. Section 2 presents the Iceberg-Cube problem and an example of it. Section 3 describes three approaches: bottom-up computation, top-down computation, and top-down computation with pruning. Section 4 presents results of testing the three algorithms on data from a student database. Section 5 concludes the paper.

2. Problem

The problem is to determine which group-bys in a given database table satisfy an aggregate condition, such as minimum support. For a three dimensional data cube this problem may be represented as [1]:

```
SELECT      A,B,C,COUNT(*),SUM(X)
FROM        R
CUBE BY     A,B,C
HAVING      COUNT(*) >= minsup,
```

where *minsup* represents the minimum support, i.e., the minimum number of tuples a group-by must contain. The result of such a computation can be used to answer any queries on a combination of the attributes (dimensions) A,B,C that require COUNT(*) to be greater than minimum support.

A	B	C	D
a1	b2	c1	d1
a2	b1	c1	d2
a2	b2	c2	d2
a3	b2	c2	d1
a3	b3	c1	d1
a4	b3	c3	d2
a5	b2	c2	d1

Figure 1: Sample Relation

Combination	Count
b2	4
b2-c2	3
c1	3
c2	3
d1	4
d2	3

Figure 2: Output

Figure 1 shows a sample relation with four attributes ABCD of varying cardinalities. Suppose the condition is that a combination of attribute values must appear in at least three tuples (*minsup* is 3). Then the output from the algorithm is as shown in Figure 2. All algorithms analysed in this paper produce the same results.

3. Approach

We consider three methods for computing Iceberg Cubes. All methods yield identical output. The output is a table that contains combinations of values for attributes that meet the specified condition, and the aggregate value for each. Figure 3 shows a 4-dimensional lattice that represents all combinations of four attributes and the relationships between these combinations.

We must determine for each combination whether or not it has a value or values that satisfy the minimum support requirement.

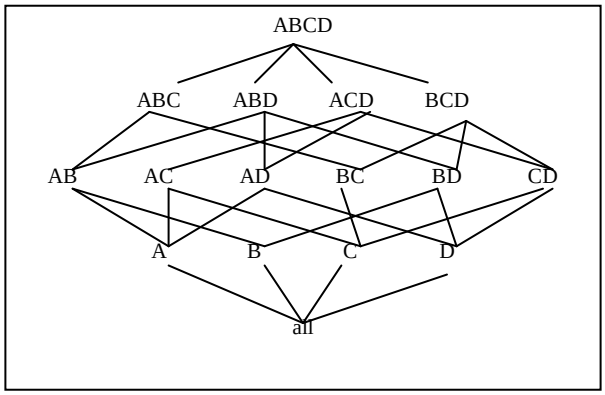

Figure 3: 4-Dimensional Lattice [1]

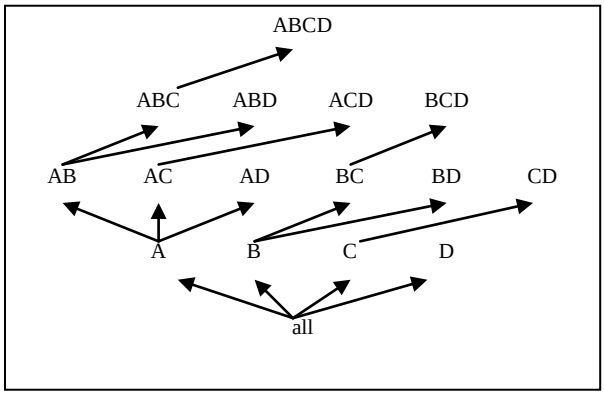

Figure 4: BUC Processing Tree

Bottom-Up Computation

The Bottom-Up Computation (BUC) algorithm is given in Figure 5 [1]. This algorithm computes the cube beginning with the smallest, most aggregated group-bys and recursively works up to the largest, least aggregated group-bys. The processing order can be seen in Figure 4, where BUC begins processing at the leaves of the tree and recursively works its way upwards. If a group-by does not satisfy the minimum support condition, then the algorithm does not recurse to calculate the next largest group-by.

The BUC algorithm begins by taking the entire input and aggregating it. The aggregate function is not described in [1] and our solution is to perform an ORDER BY of the input on attribute *d* (line 4). The input is then partitioned on attribute *d* (line 9). The Partition function is given in Figure 6. In the first call to this function, *d* is 0, so the algorithm partitions on attribute A, returning the count for each unique value of attribute A. Partition selects attribute *d* and count(*) from *inputTable* and does a group-by on that attribute (line 5). The resulting counts are then stored in an array along with the name of the temporary table that was created (lines 8 - 12). This array is then return to the BUC algorithm.

For each of these values between 0 and the cardinality of A, the count is checked to see if it is greater than *minsup*. If it is, then the value is inserted into the results table along with its count. BUC is then recursively called on the next attribute (in this case attribute B) with only the tuples containing that specific value of A as input.

When BUC is called with the table in Figure 1 as input, it first calls Partition on attribute A, producing a count for each unique value of A in the entire table. If *minsup* is 40%, or three tuples, no value of A has count > *minsup* so the algorithm returns and recurses on attribute B. Partition is called this time on attribute B. The count for $b_1$is not greater than *minsup*, so the algorithm checks the count for b_2, which is 4. The tuple <b_2,4> is inserted into the Results table. The algorithm then recurses on only those tuples which contain b_2. We now have three tuples as

```
Function BUC(inputTable, dim, start)
Inputs:
  inputTable: name of table to perform BUC on
  dim: attribute (dimension) to partition
  start: starting attribute for the itemset currently
    being counted
Global Variables:
 numDims: number of attributes in input table
 cardinality[numDims]: cardinality of each attribute
 minsup: minimum number of tuples in a partition for
    it to be output
 result: table to hold output itemsets
 fields[]: array of field names for input table
 level: represents depth of current recursive call
Local Variables:
 dataCount: first index contains name of table created
    by partition(); the remaining indexes contain
    counts for each value of attribute d
 strTemp: holds name of itemset found to be frequent
 count: holds count of value currently being examined
 curTable: holds the result of aggregating the inputTable
Method:
1.  level = level + 1;
2.  newTable = "temptable" & dim;
3.  for d = dim to numDims
4.    curTable = Run ("select * from " & inputTable &
        " order by " & fields[d]);
5.    if curTable.TupleCount <= 1 then
6.      return 0;
7.    end if
8.    C = cardinality[d];
9.    dataCount = partition(inputTable, d, C);
10.   k = 0;
11.   for i = 0 to C - 1
12.     count = dataCount[i + 1];
13.     if count >= minsup then
14.       if level = 1 then
15.         start = d;
16.       end if
17.       strTemp = curTable[k, start];
18.       for j = i - level + 2 to i
19.         strTemp = strTemp & "-" & curTable[k, j];
20.       end for
21.       Run ("insert into Results (Combination, Count)
            values (" & strTemp & "," & count  & ")";
22.       Run ("select into " & newTable & " from " &
            inputTable & " where " & fields(d) & " = " &
            curTable[0,d];
23.       ret_val = BUC(newTable, d + 1, start);
24.     end if
25.       k = k + c;
26.   end for
27. end for
28. Run ("drop table " & newTable);
29. level = level - 1;
30. return(0);
```

Figure 5: BUC Algorithm

```
Function partition(inputTable, d, C)
Inputs:
  inputTable: string containing name of table to be
    partitioned
  d: dimension on which to partition inputTable
  C: cardinality of dimension d
Global Variables:
  fields[]: array containing names of fields
Local Variables:
  count[]: array where the first position contains name of
    newTable; the remaining positions contain count of
    tuples in each partition
  strGroup: field to group by
  newTable: name of table in which group by results
    are stored
Method:
1.  int count[C];
2.  strGroup = fields[d];
3.  newTable = "temptable" & dim;
4.  Run ("select " & strGroup & ", count(*) as Count into
      " & newTable & " from " & inputTable & " group by
      " & strGroup);
5.  curTable = ("select * from " & newTable);
6.  size = 0
7.  for i = 0 to curTable.RecordCount - 1
8.    count[i + 1] = curTable[i, curTable.FieldCount - 1];
9.    size = size + 1;
10. end for
11. count[0] = newTable;
12. return count;
```

Figure 6: Partition Algorithm

input and Partition is called on attribute C for that input. The count for c_1 is only 1 so the algorithm looks at c_2. The count for c_2 is three so <b_2-c_2,3> is inserted into the Results table and we recurse on partition c_2. This time when Partition is called on D no counts are greater than *minsup* so the algorithm returns. This process continues until the Results table contains all of the combinations and counts seen in Figure 2.

To improve the running time of this algorithm, the attributes should be ordered from lowest to highest cardinality [1]. This ordering increases pruning on the first few recursions and reduces the number of recursive calls.

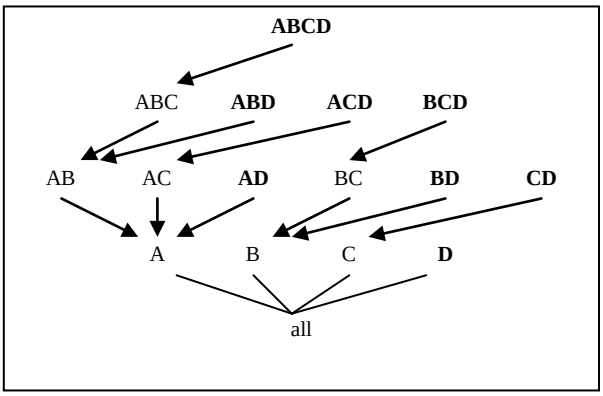

Figure 7: TDC Processing Tree

Top-Down Computation

The Top-Down Computation (TDC) algorithm is shown in Figure 8. This algorithm was devised based on an algorithm given in [3]. It uses orderings, produced by the Order algorithm in Figure 9, to reduce the number of passes through the database. With *n* attributes, Order produces 2^{n-1} orderings and calls TDC on each of these. For example, for attributes A, B, C, and D, the 8 possible orderings are ABCD, ABD, ACD, AD, BCD, BD, CD and D. When TDC is called with a specific ordering, it is not only that ordering which is counted, but also the combinations that are subsets of that ordering. For ordering ABCD, the following combinations are counted on the same pass through TDC: A, AB, ABC and ABCD. These combinations appear on the same path in the processing tree in Figure 4. Separate calls are made to TDC for the other seven orderings.

Order is a recursive function. The depth of recursion is indicated by the variable *count*. On each recursion, *arrOrder* is copied into a new array called *arrTemp* that will contain one more attribute than *arrOrder*. This extra index is then filled with the name of the next attribute to be added, *attributes*[*count*]. Order is called again for both the original *arrOrder* and the new *arrTemp*. Once *count* reaches the second last attribute, the recursion stops. All possible orderings for a given set of attributes end in the name of the last attribute, so at this point the last attribute is appended to *arrOrder*. TDC is called with *arrOrder* and its the number of attributes in *arrOrder* as arguments.

```
Function TDC(arrOrder, size)
Inputs:
  arrOrder[]: attributes to order the select statement by
  size: number of attributes in arrOrder
Global variables:
  minsup: minimum support
  numTuples: number of tuples in the database table
  results: table that contains results of TDC()
Local variables:
  arrNext[]: hold the tuple being examined
  arrHold[]: the last unique tuple examined
  strOrder: string containing ordering of fields
  i, j: counters
  curTable: temporary table that holds results of order
    by query
Method:
1.  strOrder = arrOrder[0];
2.  for i = 1 to size - 1
3.     strOrder = strOrder & "," & arrOrder[i]
4.  end for
5.  Run ("select " & strOrder & " from " & tableName & "
      order by " & strOrder);
6.  for i = 0 to size - 1
7.    arrHold[i] = curTable[0, i];
8.  end for
9.  for i = 0 to numTuples
10.   arrNext[0] = curTable[i , 0];    // first attribute of
                                       // current tuple
11.   for j = 0 to size - 1
12.     if j > 0 then
13.       arrNext[j] = arrNext[j - 1] & "-" & curTable[i , j];
14.     end if
15.     if arrNext[j] != arrHold[j] then
16.       if count[j] >= minsup then
17.         Run ("insert into results (Combination, Count)
              values (" & arrHold[j] & "," & count[j];
18.       end if
19.       count[j] = 0;
20.       arrHold[j] = arrNext[j];
21.     end if
22.     count[j]++;
23.   end for
24. end for
25. return(0);
```

Figure 8: TDC Algorithm

```
Function Order
(arrOrder, count, size)
Inputs:
  arrOrder[]: attributes that will be used for the
    ordering
  size: number of attributes in arrOrder[]
  count: number of recursions before this point
Global Variables:
  attributes[]: array of strings that represent attribute
    names
  numAttr: number of attributes
Local Variables:
  arrTemp[]: stores new ordering
  i: counter
  ret_val: return value
Method:
1.  string arrTemp[size + 1];          // make arrTemp one
2.  index bigger than arrOrder[]
3.  i = 0;

4.  if numAttr - count > 1 then
5.    for i = 0 to size - 1
6.      arrTemp[i] = arrOrder[i];
7.    end for
8.    arrTemp[size] = attributes[count];
9.    Order(arrTemp, count + 1, size + 1);
10.   Order(arrOrder, count + 1, size);
11. else
12.   for i = 0 to size - 1
13.     arrTemp[i] = arrOrder[i];
14.   end for
15.   arrTemp[size] = attributes[numAttr - 1];
16.   TDC(arrTemp, size + 1);
17. end if
18. return(0);
```

Figure 9: Order Algorithm

If Order is called on the attributes in Figure 3, then the first ordering passed to TDC is ABCD. The algorithm performs an ORDER BY on the database based on the ordering. It then visits every tuple in the resulting table to determine the count for all itemsets that are subsets of ABCD.

The variable *arrNext* is set to the value of the first tuple in the query result. The array *arrHold* stores the value of the previous tuple (it is empty of the first pass of the first tuple). For each tuple in the query result *arrNext* is set to the value of the current tuple. Beginning with attribute A, if the value of attribute A in *arrNext*[*0*] is the same as the value held in *arrHold*[*0*], then *count*[*0*] is incremented. The loop continues and checks if the values are the same for attribute B. If they are then *count*[*1*] is incremented, and so on. If the new value is different from the one stored in *arrNext*, then we check if count for that attribute is greater than *minsup* and make the appropriate insertion into the Results table. The combination to be inserted contains all values in *arrNext* from 0 to the current attribute. Following this, count is reset to 0, *arrNext* is copied to *arrHold* and the new value is stored in *arrNext* for further computation. The process continues for every ordering that Order passes to TDC.

For the table in Figure 1, $[a_1,b_2,c_1,d_1]$ is copied into *arrHold* and the attribute counts are incremented to 1. For the second pass *arrNext*[*0*] = a_2, so *count*[*0*] is checked. It is less than *minsup* (where *minsup* = 40% from the previous example), so the new tuple $[a_2,b_1,c_1,d_2]$ is copied into *arrHold* and the counts are reset to 0. No value of A has count greater than or equal to *minsup* so TDC returns with no output. The same follows for the next three orderings: ABD, ACD, and AD.

The fifth ordering is BCD. When TDC is called, an ORDER BY is done on the attributes B, C, and D. The first tuple, $[b_1,c_1,d_2]$, is copied into *arrHold* and all attribute counts are incremented to 1. The second tuple contains a different B value than the first, so the current counts are checked. They are all less than *minsup*, so they are reset to 0 and $[b_2,c_1,d_2]$ is copied into *arrHold*. On the next iteration *arrNext* = $[b_2,c_2,d_1]$, so *arrHold*[*0*] = *arrNext*[*0*] and *count*[*0*] is incremented. *arrHold*[*1*] is different than *arrNext*[*1*], so *count*[*1*] is checked. It is less than *minsup*, so c_2 is copied into *arrHold*[*1*] and *count*[*1*] is reset to 0. This process continues with no output until tuple six.

When tuple six is reached, *arrHold* = $[b_2,c_2,d_2]$ and *arrNext* = $[b_3,c_1,d_1]$. At this point *count* = [4, 3, 1], so $[b_2,4]$ and $[b_2\text{-}c_2,3]$ are both inserted into the Results table because they have counts greater than or equal to *minsup*. All counts are reset to 0 and *arrNext* is copied into *arrHold* as before.

This process continues until the current ordering is completed and the orderings BD, CD and D are also completed. When all orderings have been completed, Results holds all attribute value combinations that satisfy minimum support.

Top-Down Computation with Pruning (TDC-P)

The TDC-P algorithm adds pruning to TDC. The TDCP is given in Figure 10 in terms of the changes made to the TDC algorithm of Figure 8. Changes must also be made to the Order algorithm in Figure 9. The new OrderTDCP algorithm is given in Figure 11. An additional integer array called *frequent* is added to TDCP. This array is initialized to 0. When an itemset is found to be above *minsup*, *frequent*[*i*] is set to 1, where *i* corresponds to the last attribute in the itemset. In lines 25.1 through 25.6, *frequent* is analysed and the lowest index that still holds a 0

found. The number of attributes minus this index is returned to OrderTDCP. OrderTDCP returns *ret_val* - 1 times and then continues execution.

For example, for Figure 1 and a *minsup* of 3, on the first ordering (ABCD) no value of attribute A satisfies *minsup*. TDCP returns the number of attributes minus the index for attribute A, which is 4. OrderTDCP returns 4 - 1 times before creating any new orderings. This eliminates calling TDCP on the orderings ABD, ACD and AD. All combinations counted by these orderings include A, yet we already know that no value of A is greater than *minsup* so we can safely eliminate these three orderings. OrderTDCP continues with regular execution by creating the next itemset, BCD, and calling TDCP. The result is the same as for the BUC and TDC algorithms.

```
Function TDCP(arrOrder, size)
Local Variables:
  ret_val: integer return variable
  frequent[size]: array initially set to zero as flag for
    whether or not an attribute value was found above
    minsup

17.1  Run ("insert into results (Combination, Count)
          values (" & arrHold[j] & "," & count[j];
17.2  frequent[j] = 1;

25.1  for i = 0 to size - 1
25.2    if frequent[i] = 0 then
25.3      ret_val = size - i;
25.4      exit for
25.5    end if
25.6  end for
25.7 return ret_val;
```

Figure 10: TDCP Algorithm

```
Function OrderTDCP(arrOrder, count, size)
Local Variables:
  ret_val: integer return variable

3.1   i = 0;
3.2   ret_val = 0;

9.    ret_val = OrderTDCP(arrTemp, count + 1,
        size + 1);
10.1 if ret_val <= 1 then
10.2   ret_val = OrderTDCP(arrOrder, count + 1,
size);
10.3 end if

16.   ret_val = TDCP(arrTemp, size + 1);

18.   return (ret_val - 1);
```

Figure 11: OrderTDCP Algorithm

4. Results

The data we used to test these algorithms is grading data from the University of Regina's student record system. The original table has 59689 records and 21 attributes, most of which have cardinalities below 150. We ran tests on the original table and on 6 smaller tables which are subsets of the original table. The smaller tables contained 45000, 30000, 20000, 10000, 5000, and 2000 records. Input for each test run included one of these tables and three or more attribute names to calculate the attribute value combinations from. We call these attribute names the input attributes for a test run.

The algorithms were implemented using Microsoft Access and Visual Basic. Testing was done on a Pentium III 600 MHz PC with 256MB of RAM. We measured elapsed time on a dedicated machine. Time for input and output was included. In previous testing of BUC, the actualization of output was not implemented [1].

Test Series 1: Pruning Effectiveness of TDC-P

We first ran tests to measure the effectiveness of the pruning in TDC-P. For all input data, TDC-P ran at least as fast as TDC, and was faster for most input data. An example of the pruning effectiveness of TDC-P can be seen in Figure 12 where four attributes were used as input. The execution times for both runs of TDC-P were faster than those for TDC.
The execution time for the TDC-P algorithm relies on the ordering of the input attributes, as does the BUC algorithm [1]. This ordering is based on the cardinality of the attributes. As can be seen in Figure 12, the ordering does not significantly affect the execution time of the TDC algorithm because no pruning occurs. For four attributes as input, n is 4, so the TDC algorithm made 2^{n-1} = 8 passes over the data for both orderings. TDC visits every tuple the same number of times regardless of the order of the attributes.

Execution time differed for TDC-P when the attributes used were ordered from low to high cardinality and from high to low cardinality. High to low cardinality allows for pruning at an earlier stage and so the execution times for TDC-P were faster with this ordering. In the tests used for Figure 12, TDC-P made 7 passes over the data when the attributes were ordered from low to high cardinality, but only 5 passes were made when the attributes were ordered from high to low cardinality.

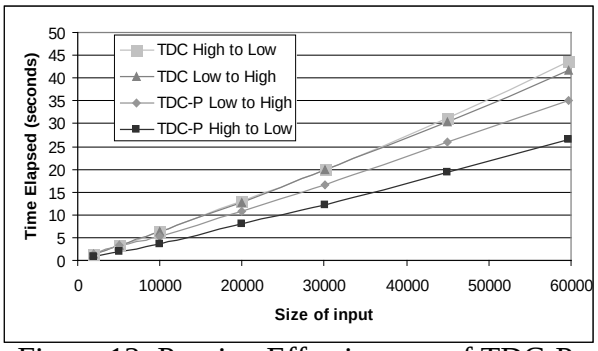

Figure 12: Pruning Effectiveness of TDC-P with Four Attributes Input

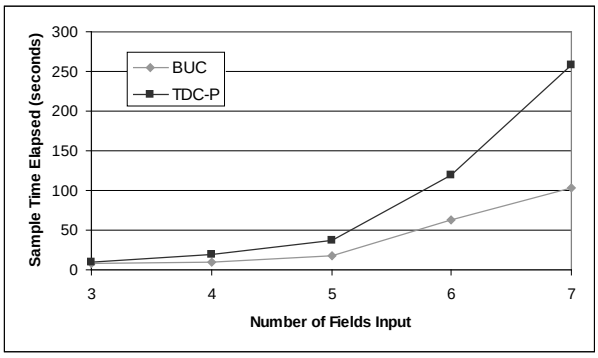

Figure 13: Varying Number of Attributes

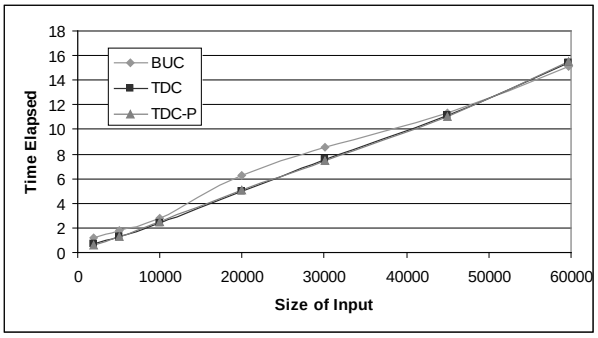

Figure 14: Three Attributes Input

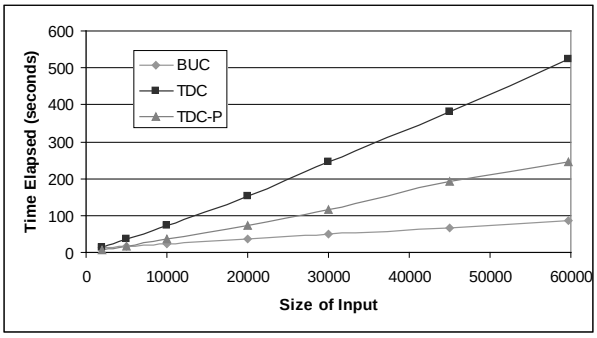

Figure 15: Seven Attributes Input

Test Series 2: Varying Number of Attributes

We tested the three algorithms with varying numbers of attributes. Figure 14 shows the results of running the algorithms with three attributes as input. All the algorithms had similar execution

times. As we increased the number of attributes input, the execution times of the TDC and TDC-P algorithms grew longer than that of the BUC algorithm. In Figure 15, the execution time of TDC-P is significantly longer than that of BUC for seven attributes. Figure 13 shows the difference in performance between BUC and TDC-P as the number of input attributes is increased. BUC performs faster than TDC-P for many input attributes.

Test Series 3: Varying Cardinality of Attributes

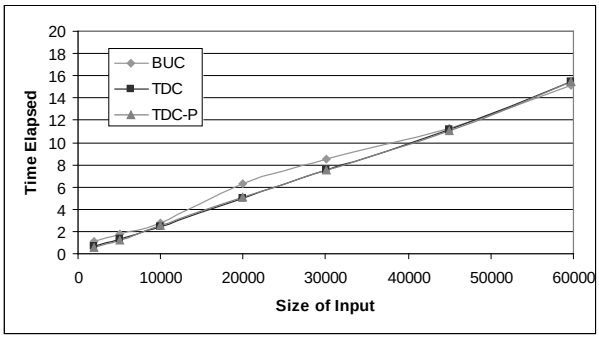

Figure 16: Three Fields - Low Cardinalities

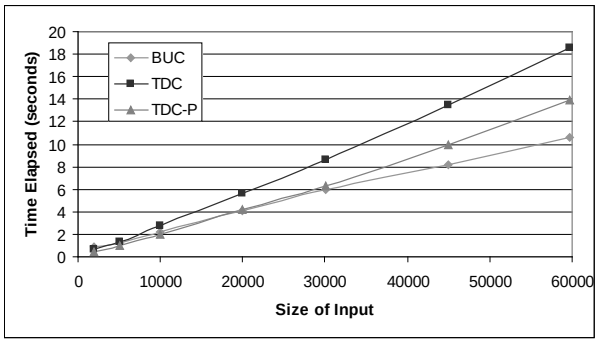

Figure 17: Three Fields - High Cardinalities

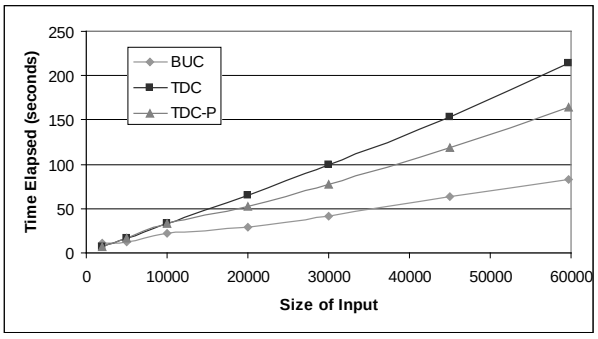

Figure 18: Six Fields - Low Cardinalities

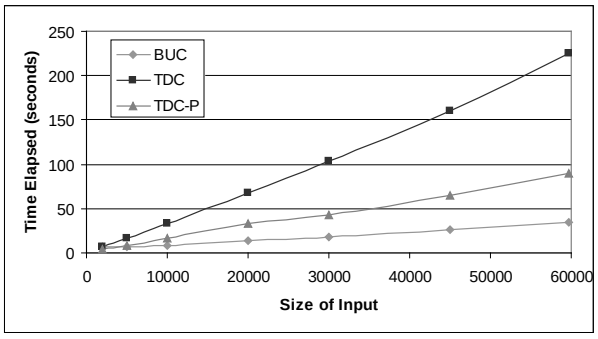

Figure 19: Six Fields - High and Low Cardinalities

We investigated the effect of varying the cardinality of the attributes on the performance of the three algorithms. This allowed us to compare the pruning capability of the algorithms. We arbitrarily chose a combination of tests using attributes with low cardinalities and attributes with high cardinalities. The ‘Pidm’ attribute had a cardinality of around 20000 for the largest of the input tables and so it offered the most likely possibility for early pruning. Only attributes with cardinalities under 150 for the largest table were used for the tests containing only attributes with low cardinalities.

Since TDC-P runs faster when it can prune at an early stage, we thought it would have its best execution times when compared to BUC when at least one attribute with high cardinality was present. Figure 16 shows the execution times when three attributes with all low cardinalities were used as input. Since the TDC and TDC-P algorithms had similar running times we know that not much pruning could have occurred. This accounts also accounts for the similarity in running times between TDC-P and BUC because BUC could not have pruned much either. In Figure 17, we included the 'Pidm' attribute and two other higher cardinality attributes. This allows TDC-P to

prune more and therefore improve its running time, but it also allows BUC to prune more. As can be seen in the chart the running time for BUC improved even more than the running time for TDC-P.

Figures 18 and 19 show the results of similar tests run with six attributes instead of three. The result of these tests is faster than the TDC-P algorithm. The execution time of TDC-P improved in Figure 19 compared to Figure 18, but it was still slower than the BUC algorithm.

5. Conclusion

We described three algorithms for the Iceberg-Cube problem, which is to identify attribute values which meet an aggregate threshold requirement. The results of our testing show that the BUC algorithm is faster than the TDC and TDC-P algorithms and produces the same output. TDC-P was more effective than TDC, but could only compete with BUC when there were very few attributes as input.

The BUC algorithm ran faster than the TDC-P algorithm because it begins with the smallest group-bys possible, which are a lot faster to compute than the largest group-bys the TDC starts with. Although BUC and TDC-P both employ pruning, by the time TDC-P begins pruning it has done a lot more work than BUC to arrive at the same point. As the number of attributes increases, this problem does also because it will take a lot longer for TDC-P to perform the first few group-bys. The conclusion is that BUC prunes earlier and more efficiently than TDC-P.

Acknowledgements

The authors would like to thank Steve Greve for his implementation of the original variant of the TDC algorithm and a copy of the database he used for his experiments.

References

[1] K. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. *SIGMOD Record*, 28(2):359-370, 1999.

[2] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing Iceberg Queries Efficiently. In *Proc. of the 24th VLDB Conference*, pages 299-310, New York, August 1998.

[3] S. Greve. *Experiments with Bottom-up Computation of Sparse and Iceberg Cubes*. CS831 Course Project, Department of Computer Science, University of Regina. Regina, Canada.

[4] V. Harinarayan, A. Rajaraman and J. Ullman. Implementing Data Cubes Efficiently. *SIGMOD Record*, 25(2):205-216, 1996.

[5] S. Sarawagi, R. Agrawal, A. Gupta. *On Computing the Data Cube*. Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.